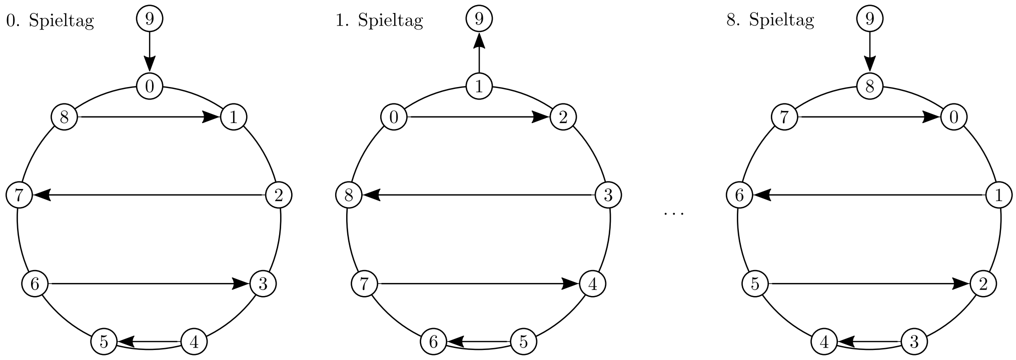
Elmar Schömer
Ann-Christin Wörl

5. Übungsblatt

Abgabe: Dienstag, der 28.11.2023, 14:00 Uhr

Aufgabe 1: Turnier (10+10+10* Punkte)

Wir wollen einen Spielplan für die Hinrunde der Bundesliga-Saison erstellen und benutzen dazu das Schema aus der folgenden Abbildung (siehe auch `https://de.wikipedia.org/wiki/Jeder-gegen-jeden-Turnier` bzw. `https://en.wikipedia.org/wiki/Round-robin_tournament`)



Im obigen Beispiel sind 10 Mannschaften gegeben, die von 0 bis 9 durchnumeriert sind. Ebenso numerieren wir die Spieltage von 0 bis 9. Die Pfeile in den Diagrammen geben an, welche Spielpaarungen (Heimmannschaft -> Gastmannschaft) am jeweiligen Spieltag stattfinden sollen.

1. Erklären Sie in Worten, wie sich das Diagramm des $(i + 1)$. Spieltages aus dem Diagramm des i. Spieltages ergibt. Beschreiben Sie mathematisch, welche Heimmannschaft h gegen welche Gastmannschaft g am i. Spieltag antritt.
2. Schreiben Sie ein Python-Programm, das die Spielpaarungen aller Spieltage ausgibt. Lesen Sie dazu aus der Datei `Bundesliga-Klubs.txt` die Mannschaftsnamen in ein Feld ein, und verwenden Sie dieses als Zuordnung der Mannschaftsnummern zu den Mannschaftsnamen.

Zusatzaufgabe:

3. Begründen Sie, warum das obige Schema funktioniert.

Aufgabe 2: Conways „Spiel des Lebens" (10+5+5 Punkte)

Wir wollen Conways „Spiel des Lebens" (siehe `https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens`) implementieren und in einfacher Form visualisieren. Als Datenstruktur für das Spielfeld benutzen wir ein 2-dimensionales Feld. Jeder Feldeintrag (im Folgenden Zelle genannt) kann nur zwei verschiedene Werte annehmen: '.' für tot und 'o' für lebendig. Es gibt vier Regeln, wie sich die Belegung des Spielfeldes von einer Generation zur nächsten verändert:

Regel 1: Wenn eine tote Zelle genau drei lebende Nachbarzellen hat, wird sie zum Leben erweckt.

z.B.

·	·	·
o	·	·
·	o	o

⇒

	o	

Regel 2: Eine lebende Zelle mit weniger als zwei lebenden Nachbarnzellen stirbt.

z.B.

·	·	·
·	o	·
·	·	o

⇒

	·	

Regel 3: Eine lebende Zelle mit zwei oder drei lebenden Nachbarnzellen bleibt am Leben.

z.B.

·	o	·
o	o	·
·	·	o

$\Rightarrow$

	o	

Regel 4: Eine lebende Zelle mit mehr als drei lebenden Nachbarzellen stirbt.

z.B.

o	o	o
·	o	o
o	·	·

$\Rightarrow$

	·	

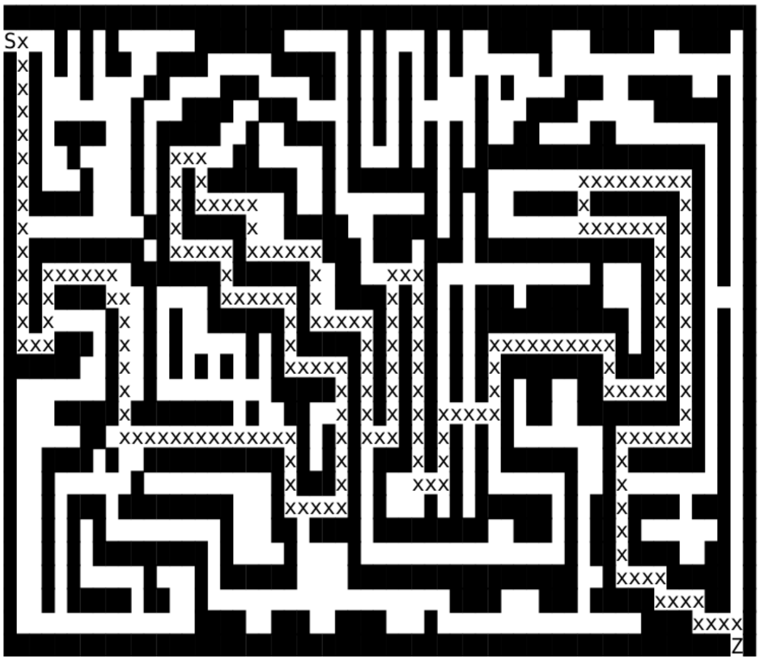
Das folgende Python-Programm visualisiert die Belegung eines zeitveränderlichen 2-dimensionalen Feldes aus den Symbolen '.' und 'o'. Verwenden Sie dieses Programm als Grundlage für die Simulation und Visualisierung des oben beschriebenen „Spiel des Lebens“.
Tipp: Verwenden Sie zwei 2-dimensionale Felder - eines für die Generation zum Zeitpunkt t und eines für die zum darauffolgenden Zeitpunkt $t+1$.

1. Betrachten Sie ein Spielfeld mit Rand.
2. Betrachten Sie ein Spielfeld ohne Rand, indem der linke mit dem rechten Rand und der obere mit dem unteren Rand identifiziert wird, so dass ein Torus entsteht.
3. Beobachten Sie, wie sich ein "Gleiter" bzw. ein "f-Pentomino" verhalten. Interessant ist auch die Entwicklung einer zufällig erzeugten Startkonfiguration.

```python
import time, IPython.display
#import sys,random

m = 26 # Zahl der Zeilen
n = 100 # Zahl der Spalten

G = [ ['.' for s in range(n)] for z in range(m)]

# Gleiter
G[0][1] = 'o'
G[1][2] = 'o'
G[2][0] = 'o'
G[2][1] = 'o'
G[2][2] = 'o'

# f-Pentomino
G[m//2-1][n//2] = 'o'
G[m//2-1][n//2+1] = 'o'
G[m//2][n//2-1] = 'o'
G[m//2][n//2] = 'o'
G[m//2+1][n//2] = 'o'

for i in range(100):
    time.sleep(0.1)
    IPython.display.clear_output(wait=True)

    for z in range(m):
        tmp = G[z][n-1]
        for s in range(n-1,0,-1):
            G[z][s] = G[z][s-1]
        G[z][0] = tmp

    for z in range(m):
        for s in range(n):
            print(G[z][s],end='')
        print()
    #sys.stdout.flush()
```

Aufgabe 3: Labyrinth (5+5+5+10+5+10* Punkte)

Gegeben sei ein Labyrinth als zweidimensionales Feld in der Datei `data/labyrinth.txt`. Der Eingang ist mit dem Buchstaben 'S' und der Ausgang mit dem Buchstaben 'Z' markiert.



1. Schreiben Sie eine Funktion zum Einlesen der Textdatei. Diese Funktion soll als Argument den Namen der Datei erhalten und als Rückgabewert ein zweidimensionales Feld von Zeichen liefern.

2. Schreiben Sie eine Funktion, die als Argument ein zweidimensionals Feld von einzelnen Zeichen erhält und diese auf dem Bildschirm mit Hilfe der print-Anweisung ausgibt.

3. Schreiben Sie eine Funktion, die als Argument ein zweidimensionales Zeichenfeld L und ein einzelnes Suchzeichen c erhält und eine Position (z,s) als Tupel ermittelt, so dass $L[z][s] = c$. Falls keine Feldposition mit dieser Eigenschaft existiert, soll das Tupel $(-1, -1)$ zurückgegeben werden.

4. Schreiben Sie eine Funktion, die auf dem zweidimensionalen Zeichenfeld eine Breitensuche von einer Startposition (z_0, s_0) durchführt. Der Rückgabewert soll ein zweidimensionales Feld von Vorgängerverweisen für den Baum der kürzesten Wege sein. Das heißt, für jedes von der Startposition erreichbare Feld (z_k, s_k) steht in $pred[z_k][s_k] = (z_{k-1}, s_{k-1})$ die Position des Vorgängerfeldes auf einem kürzesten Weg von (z_0, s_0) zu (z_k, s_k).

$$(z_k, s_k) \leftarrow (z_{k-1}, s_{k-1}) \leftarrow \ldots \leftarrow (z_0, s_0)$$

   (**Tipp:** Das Labyrinth kann man als einen ungerichteten Graphen $G = (V, E)$ auffassen, bei dem die Knoten V den freien Feldern des Labyrinths entsprechen und zwei Knoten durch eine ungerichtete Kante verbunden sind, wenn die zugehörigen freien Felder benachbart sind. Es ist jedoch nicht notwendig, eine Graph-Datenstruktur in Form von Adjazenzlisten aufzubauen. Vielmehr kann man die Breitensuche direkt auf dem 2-dim Zeichenfeld L ausführen, wenn man für die 'besucht'-Markierung und das Vorgängerfeld 'pred' ebenfalls 2-dim Felder anlegt und in der Warteschlange Q 2-Tupel (z, s) von Feldpositionen verwaltet.)

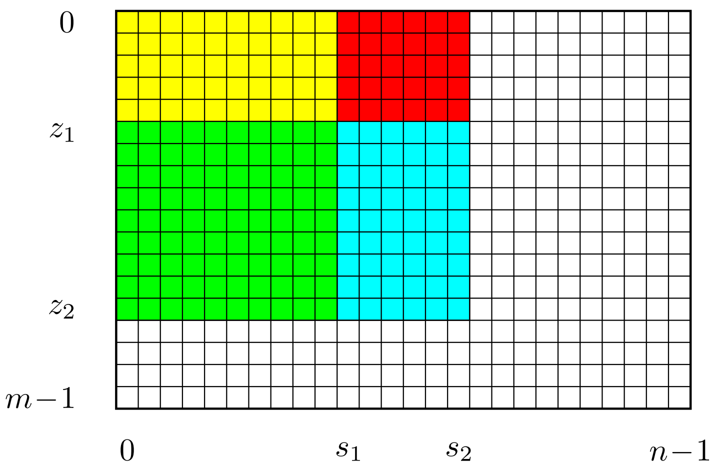
5. Schreiben Sie eine Funktion, die einen kürzesten Weg vom Eingang zum Ausgang des Labyrinths in das 2-dim Zeichenfeld in Form von 'x'-en einträgt.

   Zusatzaufgabe: Visualisieren Sie die Breitensuche im zeitlichen Verlauf, so dass erkennbar wird, welche Labyrinth-Felder nacheinander besucht werden (ähnlich zu Aufgabe 3).

Aufgabe 4: Matrixsummation (5+5+5+5+10 Punkte)

Gegeben sei eine Matrix $\mathbf{A} \in \mathbb{Z}^{m\times n}$. Wir wollen eine schnelle Methode entwickeln, um die Summe der Elemente der Untermatrix (türkises Rechteck) der Zeilen z_1 bis z_2 und der Spalten s_1 bis s_2 zu berechnen:

$$S(z_1, z_2, s_1, s_2) = \sum_{k=z_1}^{z_2} \sum_{l=s_1}^{s_2} A_{kl}$$



Später wollen wir für viele verschiedene Werte $0 \leq z_1 \leq z_2 < m$ und $0 \leq s_1 \leq s_2 < n$ die Summe $S(z_1, z_2, s_1, s_2)$ ausrechnen. Deshalb führen wir einen Vorverarbeitungsschritt durch, indem wir eine Matrix $\mathbf{C} \in \mathbb{Z}^{m\times n}$ berechnen.

$$C_{ij} = \sum_{k=0}^{i} \sum_{l=0}^{j} A_{kl}$$

1. Zeigen Sie, dass für $0 < z_1 \leq z_2 < m$ und $0 < s_1 \leq s_2 < n$ das Folgende gilt:

   $$S(z_1, z_2, s_1, s_2) = C_{z_2,s_2} - C_{z_2,s_1-1} - C_{z_1-1,s_2} + C_{z_1-1,s_1-1}$$

   Das heißt, wir können $S(z_1, z_2, s_1, s_2)$ mit nur 3 Additionen/Subtraktionen berechnen und müssen nicht alle Elemente der Submatrix anschauen - vorausgesetzt wir haben die Matrix $\mathbf{C}$ bereits bestimmt.

2. Wieviele Additionen benötigt man, wenn man alle $m \times n$ Elemente der Matrix $\mathbf{C}$ auf naive Weise berechnen möchte?

   Wir wollen nun die Elemente der Matrix $\mathbf{C}$ möglichst schnell berechnen. Dazu zerlegen wir die doppelte Summation zur Berechnung von C_{ij} in zwei einfache Summationen:

   $$C_{ij} = \sum_{k=0}^{i} \sum_{l=0}^{j} A_{kl} = \sum_{k=0}^{i} B_{kj} \quad \text{mit} \quad B_{kj} = \sum_{l=0}^{j} A_{kl}$$

3. Beschreiben Sie in Worten, wie die Hilfsmatrix $\mathbf{B} = (B_{kj})_{0\leq k<m, 0\leq j<n} \in \mathbb{Z}^{m\times n}$ aus der Matrix $\mathbf{A}$ entsteht und wie $\mathbf{C}$ aus $\mathbf{B}$ berechnet werden kann. Wenn man zuerst die Hilfsmatrix $\mathbf{B}$ und dann die Matrix $\mathbf{C}$ geschickt ausrechnet, benötigt man wesentlich weniger Berechnungsschritte im Vergleich zur naiven Methode. Begründen Sie dies!

4. Schreiben Sie ein Python-Programm, das die Matrix **A** von der Datei `A.txt` einliest und daraus die Matrix **C** berechnet und in die Datei `C.txt` schreibt. Für die Dimension der Matrizen gilt: $n = m = 1000$. Die Datei `A.txt` besteht aus n Zeilen und in jeder Zeile stehen m integer-Werte, die durch Leerzeichen voneinander getrennt sind.

5. Für eine vorgegebene Konstante K suchen wir nun eine kleinste quadratische Submatrix von **A**, so dass $S(z_1, z_2, s_1, s_2) \geq K$. Das heißt, dass $z_2 - z_1 = s_2 - s_1$ so klein wie möglich sein soll. Suchen Sie in `data/A.txt` eine solche Submatrix für den Wert $K = 314159265$.

   Tipp: Mit Hilfe einer binären statt einer linearen Suche kann man eine bessere Laufzeit erzielen.